# An Object model for engineering design

G.T. Nguyen

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

*1 9 9 2*

25*ème anniversaire*

**N° 1653**

***Programme 3***
*Intelligence artificielle, Systèmes cognitifs et Interaction homme-machine*

# AN OBJECT MODEL FOR ENGINEERING DESIGN

**Gia Toan NGUYEN**
**Dominique RIEU**
**José ESCAMILLA**

GROUPE DE RECHERCHE
**GRENOBLE**

**Avril 1992**

*RR-1653*

# Un modèle à objets pour la Conception Assistée

# An Object Model for Engineering Design

Gia Toan Nguyen, Dominique Rieu, José Escamilla

IRIMAG
Laboratoire de Génie Informatique
IMAG-Campus
BP 53 X
38041 GRENOBLE Cedex
France
Tel : (33) 76.51.45.75
e-mail : nguyen@imag.fr
fax : (33) 76.44.66.75

**Abstract.** Applications requiring sophisticated modeling techniques raise challenging issues to software designers. CAD/CAM and genetics are example of applications that call for powerful modeling techniques. Existing approaches seem limited in their ability to support their demands. Relational database systems for example support only simple tables. The need to enhance their capabilities led to non-normalized relational data models. Object-oriented programming languages and databases propose new solutions to the problems of complex and composite object modeling and manipulation. Yet, severe shortcomings impede their practicability, e.g., their inability to model multiple object representations and complex semantic relationships.
This paper is an informal overview of a data model called SHOOD based on object-oriented concepts and frame-based knowledge representation. SHOOD implements sophisticated features, such as: • object persistence, multi-methods along a specific specialization hierarchy (which is independent of the class hierarchy), •sophisticated semantic relationships, e.g., dependency relationships between objects (which are totally independent of the composition relationship), • multiple object representations, allowing the users to manipulate the objects from several points of views simultaneously, • the systematic use of a powerful meta-object kernel, which is used to implement a reflexive architecture. The paper focuses on the last two issues.

**Résumé.** Les applications qui requièrent des techniques évoluées de modélisation posent des problèmes nouveaux aux développeurs de logiciel. La CAO et la génétique en sont des exemples. Les modèles existants sont relativement limités face à leurs besoins. Le modèle relationnel gère de simples tables. Des modèles non-normalisés sont donc apparus. Les modèles à objets utilisés dans les langages de programmation et les bases de données proposent des solutions nouvelles pour la gestion et la manipulation d'objets complexes ou composites. Mais des restrictions sévères en limitent toutefois l'intérêt : absence de représentations multiples et relations sémantiques par exemple.
On propose ici un panorama informel du modèle SHOOD. Il est basé sur des techniques d'objets et de représentation de connaissances. SHOOD offre des fonctionnalités intéressantes comme : la persistence, des multi-méthodes gérées dans un graphe particulier, indépendant du graphe des classes, des relations sémantiques entre objets (relations de dépendance indépendantes des relations de composition par exemple), des représentations multiples d'objets, et un niveau méta utilisé pour implanter une architecture réflexive et extensible. On insiste sur les deux derniers points.

## 1 Introduction

SHOOD is an object-oriented data model designed to support highly dynamic applications. Design, genetics and weather forecast are examples of such applications. The challenge there is to merge partial amounts of information into a consistent and flexible computerized model.
Yet, SHOOD is not just another exotic data model. It includes features that are seldom found simultaneously in existing data models. Some of its salient characteristics are:
• the support for evolving data, both in their definitions and values,
• the support for flexible user defined semantic relationships between objects, e.g., composition relationships, specific dependency relationships (existential, exclusive, etc.),
• the support for multiple object representations, which can be concurrently defined by different users on the same objects, e.g., hydraulic and electric points of view for an engine,
• object persistence, although SHOOD does not currently provide all the functionalities of a complete database management system, e.g., concurrency control and restart facilities,
• multiple inferencing capabilities, allowing for several alternatives to be defined for attribute computations, e.g., depending on the arguments' availability.
From a historical perspective, SHOOD was designed to support the requirements of mechanical and VLSI CAD/CAM applications [22, 28]. It elaborates on knowledge representation techniques found in AI and data models found in the database area [19]. It also imports recent advances in object-oriented databases and programming languages [10, 15]. It includes such notions as meta-classes, methods, inferences, and dependencies. They are merged in a powerful and flexible data model that is currently tested on full size applications in mechanical engineering design. SHOOD is implemented in Le_Lisp™ on Sun SPARCstation™. A user-friendly interface is built on top of the system to provide an easy access to its functionalities. It is based on a menu-driven windowing system implemented in Aida™, a graphic interface development tool running under X-Window.
The paper is organized as follows. Section 2 is an in-depth analysis of the design requirements for SHOOD. Section 3 describes the meta-object kernel. It includes the meta-classes that are used to implement the concepts in the model. Section 4 is an overview of the multiple representations supported for the user objects, with indications on the classification mechanism available. Section 5 discusses research issues. Section 6 is a conclusion.

## 2 Rationale for the design of SHOOD

### 2.1 Extensibility

Among the issues raised by design applications, the evolution of objects values and structure - grossly speaking database evolution - is probably the most challenging. It questions one of the most securing aspects of data storage and manipulation, i.e., object stability. Because design applications are constructive, they require the definitions of the objects to evolve over time. This departs dramatically from business applications that manipulate large amounts of information with somewhat few data definitions. Design in contrast manipulate smaller amounts of data because the goal is to define artifacts rather than query the database.
The need for flexible data models is therefore fundamental. The commonly accepted relational model of data is for this matter all but flexible. User relations flatten object attributes that are distributed in multiple pieces among relations without explicit semantic relationships.
Recent advances in object-oriented data modeling have opened new perspectives for the definition and manipulation of complex composite objects. Besides their ability to encapsulate and reuse existing definitions - that are of first importance in design - they benefit from research in the field of object-oriented database systems [30, 31].
Yet, severe shortcomings limit their use in design applications. First there is a crucial lack of design methodologies for object-oriented systems' applications. Also, they suffer from inherent limitations in their ability to model user defined semantic relationships [8, 11]. Specific dependency relationships need to be defined between parts of a composite object, e.g., there is an

existential dependency between the airframe and an aircraft built around it, but not between the engines and the aircraft that uses them, as well as attribute propagation: the color of some airframe parts of a red aircraft must be red.
Further, fundamental concepts in object-oriented paradigms such as encapsulation, message passing, instantiation and method selection are not always well-suited for design environments. For example, encapsulation must be broken because it is usually necessary for the designers to work on the object definitions. Instantiation also requires an instance to belong to exactly one class, whereas designers require the instances to belong simultaneously to several points of views. Methods are usually attached to one class definition, which commands method selection whatever the designer intent may be.
To overcome such limitations without throwing away the object-oriented approach, we decided to define and implement an extensible object-oriented data model. This means that SHOOD provides a limited number of concepts with which the designers may customize their application model. To achieve this goal, a powerful and flexible meta-object kernel is provided. The model is reflexive and meta-circular, i.e., it is self-descriptive and is implemented using its own concepts [6]. Thus, classes are instances of generic meta-classes, class attributes are also instances of generic meta-classes describing attributes. This approach is used systematically, allowing specific classes and attributes to be defined by a single instantiation mechanism, provided the appropriate meta-classes are added to the meta-object kernel by the designers. For example, methods, inferences and constraints are also defined by instances of meta-classes. Key attributes and optional attributes are similarly defined by instances of appropriate meta-classes. This is detailed in Section 3.

## 2.2 Object evolution

In our approach, the extensibility of the object model is the support for object evolution. We believe that a powerful, flexible and maintainable system can only be achieved this way, rather than adding specific functionalities to a limited or outdated data model, in an everlasting process.
Reuse and redesign being standard in design applications, the model - when required - still benefits from the outstanding advantages of the object-oriented approach, e.g., encapsulation, sharing, inheritance, etc. It also overcomes their most stringent assumptions. For example, object instances may belong simultaneously to several classes, providing a straightforward implementation of multiple points of view [21]. Objects' instances may also be partially incomplete and inconsistent [23]. Therefore, instances are not necessarily precise representatives of specific classes. These last two features depart from fundamental assumptions in existing object-oriented models. They are necessary in design applications because the definition of an adequate object structure, even prior to any valuation of its attributes, is what engineering design is all about.
The support for incomplete and inconsistent objects is therefore the basis for object evolution [22]. It is complemented by a classification mechanism and a migration facility. Both implement the assistance given to the designers when asking: "To what object definition(s) does this partial design correspond best?" and "Attach this partial design to the most suited existing definition(s)". The classification mechanism searches in the class graph the most resembling classes for the given instances, depending on specific conditions, e.g., attribute values. The similarity is based on the degrees of completeness and consistency of the instances with respect to the class definitions. The migration facility allows the designers to override the classification mechanism and force instances to belong to less specific classes than those resulting from the system's classification process. This emphasizes user's authority over the system's automatic assistance. The modifications of the object instances can be monitored by the designers with the help of the system, which may track automatically the evolution of the objects being modified.
Another facility provided by SHOOD is a powerful inferencing capability, allowing object to be computed and updated using other objects as arguments to appropriate methods. Multiple inferences may be defined for an attribute, and may be triggered independently depending on the arguments' availability. This will be the basis for a powerful method combination mechanism [13].

## 2.3 Semantic relationships

A critical issue in design application is the support for user-defined relationships between objects. Whereas many object-oriented systems now provide some form of composite objects [14, 15], an open question is the implementation of versatile semantic relationships. Semantic networks and knowledge representation languages like SRL [9] provide powerful mechanisms for this aspect. Clearly, the usual inheritance relationship is not sufficient to model complex artifacts. Several proposals enhance their capability to support dependency links and attribute propagation, sometimes called "selective inheritance" [4]. SHOOD implements a systematic approach to the concept of relationship by allowing user-defined relationships to be defined through the meta-object level. A semantic relationship - say, an existential dependency between a composite object and one of its components - is modeled by a class that is an instance of a generic "relationship" meta-class. The particular semantics of the relationship is attached to the relationship class, i.e., in the (user-defined) methods made available to it. An in-depth analysis of the relationships in SHOOD is given in [8]. It will not be detailed any further in this paper.
A particular form of semantic relationship given for free with SHOOD is the disjoint class relationship. It states that if two classes A and B are declared disjoint, then no instance of A or any sub-class of A, may be simultaneously an instance of B or any sub-class of B. This helps clarifying the class graph for applications involving many classes. It also simplifies solving some inheritance conflicts: attributes with similar names but different semantics defined in disjoint classes cannot cause conflicts. Last, it speeds up the classification of instances in the class graph (section 4.4).

## 2.4 Evolving applications

A challenging issue is that SHOOD is designed to support not only evolving application requirements, sometimes due to coarse modeling or rapid prototyping, but also evolving knowledge on the application domain: this is a more general, more challenging goal. Example can be found in genome sequencing. In such applications, there is a need to integrate partial pieces of information whose interactions are not known in advance and whose structure may evolve over time. This is the case when refined knowledge is acquired after a previous application model has been implemented. Thus, particular forms of knowledge acquisition must be supported: the discovery of new relationships between objects, new object structures and attributes, or new inferences between object values. This calls for powerful and flexible modeling concepts. The approach adopted by SHOOD is to provide a sophisticated meta-object kernel that is used to generate the appropriate application concepts when required. This is described in Section 3. This approach is used throughout the literature to implement reflexive architectures [6].

# 3 The meta-object kernel

The meta-object kernel is the core of the system from both a conceptual and implementation point of view. As a conceptual framework, it is used as a foundation for extensibility of the model. As an implementation tool, it forms the bootstrap of the model SHOOD itself. It serves the purpose of generating the whole model from a reduced set of meta-classes. This approach bears similarities with ObjVlisp and CLOS [6, 13].

## 3.1 Meta-classes

All the classes defined in SHOOD are instances of meta-classes. The latter defines the structure and behavior of the classes. For example the class Aircraft is an instance of the meta-class Meta as depicted in figure 1. The meta-class Meta defines the minimal attributes and behavior of

classes. It includes the attributes "instance_of", "class_name", "super", "sub_classes" "instances" and "attributes".

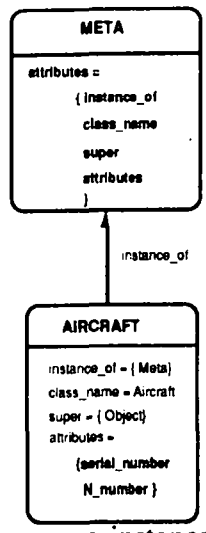


**Fig. 1.** Classes are instances of Meta.

Note that classes and instances bear an attribute called "instance_of" the domain of which is a set of classes. The value of the attribute "instance_of" is the set of classes to which the class or instance belongs. This means that the model SHOOD allows instances to belong to several classes simultaneously. This is called here multiple instantiation. This departs from usual object-based models where instances are allowed only a single class membership [12]. As will be explained in Section 4, this allows a straightforward implementation of multiple object representations.

An attribute "instances" is also defined for classes. It is the inverse of the attribute "instance_of". It gives the set of instances currently attached to the class or meta-class considered. This attribute is automatically maintained by the system. It is not depicted in figure 1.

An attribute "class_name" defined in Meta allows for classes to be referenced by a unique name.

The attribute "super" defines the set of super-classes of a class. The model therefore supports multiple inheritance. A specific class named Object is the default super-class for all objects. The super-classes are not ordered. The domain of the attribute "super" is therefore a set of classes. Sets are enclosed by "{" and "}" in the figures. Name conflicts for attributes are addressed in Section 3.3.

The attribute "sub_classes" is the inverse of "super". Sub-classes implement the inheritance relationship. The semantic of inheritance in SHOOD is structural and set inclusion. A class inherits the attributes of its super-classes. It can define new attributes. It can also refine inherited attributes. But due to the set inclusion semantics, it cannot override inherited attribute definitions. The set of instances of a class belongs to the intersection of the sets of instances of its super-classes.

Meta-classes can generate classes. Therefore, all meta-classes must inherit the structure and behavior of Meta, i.e., they must be direct or indirect sub-classes of Meta (Figure 2).

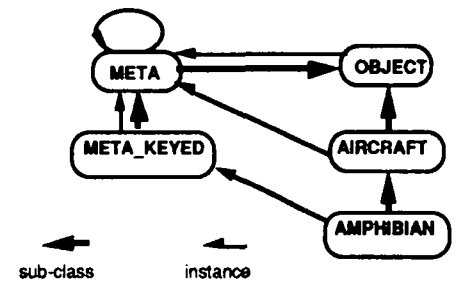


**Fig. 2.** Examples of inheritance and instantiation graphs.

Classes generate instances. They must therefore instantiate the minimal properties of classes defined in Meta, e.g., "class_name", "super". They must also define their specific attributes by instantiation of "attributes" in Meta. All classes must therefore be instances of Meta or, by set inclusion, of one of its sub-classes.
Meta-classes are classes generating particular instances, i.e., the application classes. They must therefore inherit the minimal attributes and behavior of classes, e.g., "attributes" defined in Meta. All meta-classes must therefore be sub-classes of at least one meta-class. Being classes, they must also be instances of at least one meta-class. An example in given in Section 3.2 for the meta-class Meta_keyed.
The class Object defines the minimal properties of instances, e.g., their object identifier (oid). All instances, including classes that are instances of the meta-classes, are therefore direct or indirect instances of the class Object. A consequence is also that Meta is a sub-class of Object. Further, Object being a class, it is an instance of Meta. Similarly, Meta is an instance of itself. The single exception to this approach is that Meta is a sub-class of Object, which is not a meta-class.
A consequence of inheritance in SHOOD is that classes inherit the structure and behavior of their super-classes. They must therefore be instances of the meta-classes of their super-classes. For example, the meta-classes of the class Amphibian must be the same as, or sub-classes of, the meta-classes of Amphibian's super-classes, i.e., Aircraft (Figure 2).

### 3.2 Object identification

In object-oriented systems, objects are associated with a unique system-wide identifier usually known as "oid". For user convenience, SHOOD supports also an explicit notion of "key", which is user-defined. It is a list of attributes in a class defined as an access key to object instances. It is implemented as a one-to-one mapping between key values and oids. Several keys may be defined for a class, e.g., serial number and registration number (N_number) for an aircraft. Figure 3 gives an example with the Amphibian class that has a single key formed with one attribute, i.e., "serial_number". Lists are enclosed by "(" and ")" in the figures.

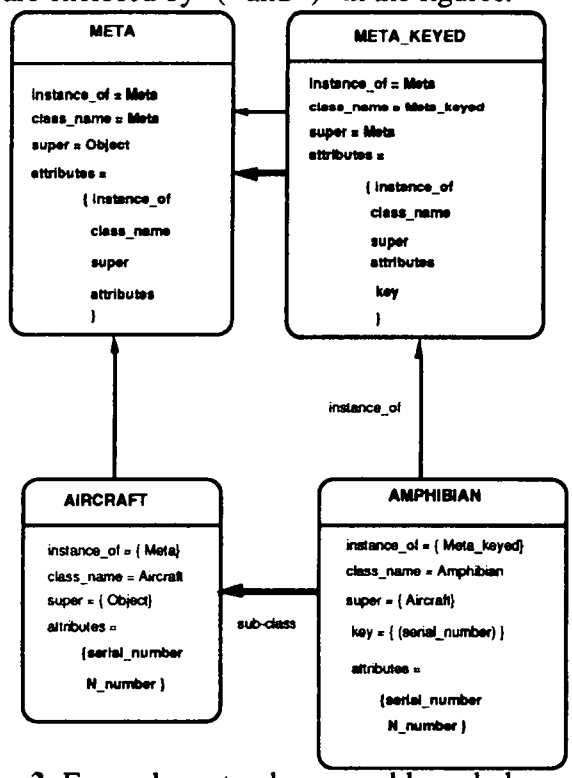


**Fig. 3.** Example meta-classes and keyed class.

Using the meta-object kernel, the structure and behavior of keyed classes are defined in a specific meta-class called Meta_keyed. It is a sub-class of Meta. As such, it inherits the minimal properties of classes. It also defines a new "Key" attribute which domain is a set of keys, i.e., of lists of key-attributes. This "Key" attribute is valued by the keys of the class instances. Meta_keyed is also an instance of Meta. As such, it values the minimal properties of classes.

### 3.3 Meta-class instantiation

**Attributes.** Class attributes are modeled in SHOOD by a specific meta-class called Meta_attribute. This is why the Section "3.3 Meta-class instantiation" includes this section (Attributes). An attribute is defined in SHOOD by a name and a set of attribute descriptors. They include a type descriptor as well as inference and constraint descriptors. This section is concerned only with type descriptors. The inference and constraint descriptors are described in Section 3.4.
Name conflicts arising from multiple inheritance are solved by using complete attribute names. A complete attribute name is a triplet <class_name, attribute_name, origin_class>, where "class_name" is the name of the class using the attribute "attribute_name", as defined by the class "origin_class". The class where an attribute is defined first is called the origin class. The attribute is inherited by all the sub-classes. Two attributes with the same name but with different origin classes are considered different. All attributes, whatever their origin class, are inherited. A partial example of attribute definition in the meta-object kernel is given in figure 4.
Type descriptors for attributes include a mandatory part and an optional part. The mandatory part is a domain definition. It can be one of: "a", "set_of" and "list_of" key-word, followed by a class name. For example the "powerplant" attribute in the class Aircraft can be defined as a "set_of Engine", where the Engine class is defined elsewhere. The optional part is a set of domain restrictions. They allow specific enumerations of values, intervals and excluded values to be specified.
Attributes are modeled by classes defining their name, origin class, domain, restrictions, constraints and inferences. They are instances of the Meta_attribute class. Giving a value to the attribute A of an instance X in class C is therefore amenable to the creation of an instance Y in the class B modeling the attribute A. The instance X refers to its attribute's value for A by the oid of Y in the class B.
If a sub-class C' of C defines a refined attribute for A, say A', then a sub-class B' of B is automatically generated. Refining an attribute is therefore a specialization process for the class defining the attribute.
A particular form of attribute declaration is the "deferred attribute" definition. It bears some similarities with the notion of deferred feature in Eiffel [18]. It restricts the multiple declarations of an attribute in the sub-graph starting at the class where the deferred declaration takes place. In all the classes of the sub-graph, all the occurrences of an attribute with the same name refer to the deferred declaration. There is therefore no other possible semantics for an attribute with the same name in that sub-graph. This helps solving some inheritance conflicts due to syntactic and semantic ambiguities - in French for sure.

**Methods.** Methods are defined outside classes. In contrast with the original object-oriented approach, e.g., Smalltalk, but like recent systems like CLOS, SHOOD relies on a specific graph for methods, which is independent of the class inheritance graph. The organization of the method graph is based on all the arguments' types. Methods are modeled by classes. A class defining a method is an instance of a particular meta-class called Meta_method. The attributes of the class are the "code" that implements the procedure and its arguments. Method overloading is implemented by defining sub-classes in the method graph. Method selection first generates an instance of the method class, then classifies it in the method graph. Currently, the first selected method is returned. Extensions are planned to consider multiple answers to the method selection process.

Consequently, method selection is based on all the arguments, not only on one selector, as implemented in most object-oriented environments.

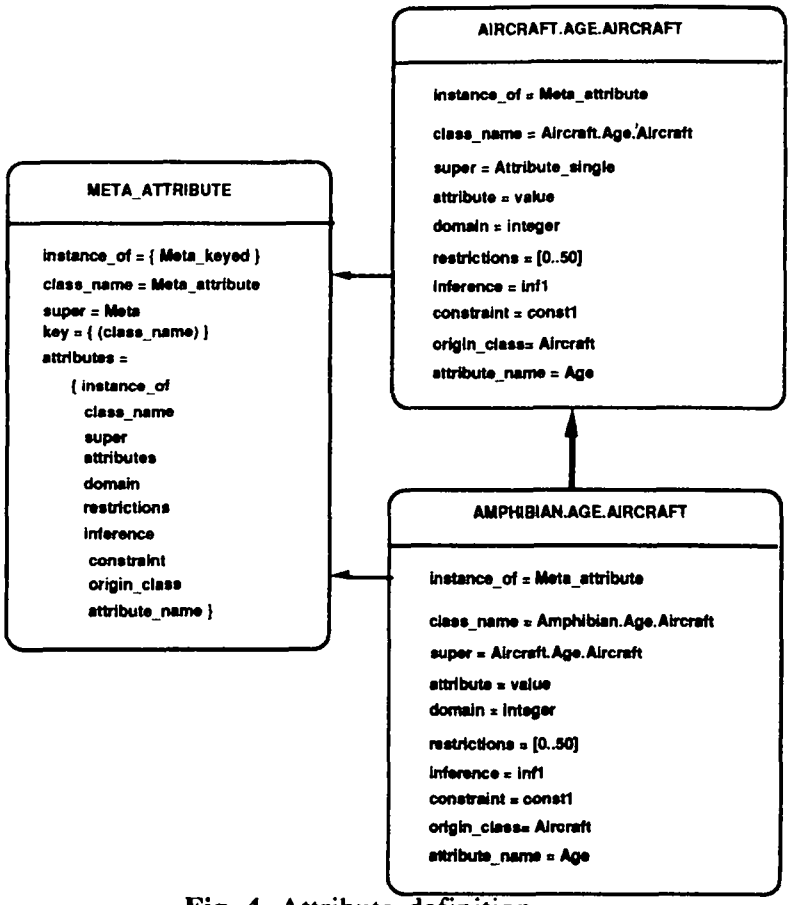


**Fig. 4.** Attribute definition.

## 3.4 Inferences and constraints

Methods are used by application programs and user interfaces to manipulate the objects. They are also used by the inferences and constraints defined in the attribute descriptors.
Inferences may be defined to compute attribute values. Several inferences may be defined for one attribute. They are ordered by default by their order of appearance in the descriptor. They are executed by order of appearance in the list until an executable one succeeds. An executable inference is one for which all arguments are instantiated. If all the executable inferences abort, an error is returned. A particular inference called "user" is provided, that can end the inference list. If all inferences abort and the "user" inference is defined, the attribute value is asked to the user.
Constraints may be defined on attributes to restrict their values. Several constraints may be defined for an attribute. They call methods. Constraints are considered mandatory, i.e., they cannot be violated. Extensions are planned to authorize optional constraints that instances may violate under user control. This is interesting for design applications because objects may often violate temporarily constraints upon user manipulations, although they are dedicated to the current class. They could be treated as exceptions, because they do not strictly follow the class definitions. But this would contradict the set inclusion semantics of inheritance in SHOOD. They are therefore explicitly considered as incomplete and/or inconsistent objects. The model supports these objects by maintaining consistency and completeness degrees for every instance in the database. Monitoring them allows to keep track of object evolution and can be smoothly integrated in the versioning mechanism.

## 4 Multiple object representations

A crucial issue in design applications is the management of multiple object representations. Cooperative work by which different designers work on complementary fields to achieve the global design project is the rule. For example, aircraft design involves aerodynamics engineers to design the airframe, mechanics to design the engines, electronics specialists to design the monitoring and "fly-by-wire" systems, hydraulics specialists, etc. Each works on particular representations of the aircraft. The various representations coexist simultaneously and contribute to the design process.
Existing object-oriented systems usually lack the powerful support for multiple object representations [26, 29]. A fundamental requirement of software engineering environments has emphasized the need for several concurrent representations of software modules: source code, compiled code, executable modules, versions, etc. This has given rise to specific implementations and proposals [1, 2]. They usually call for explicit concepts, e.g., adding new notions to a data model. For example, "representation relationships" have been proposed in the literature [4], "roles" [25] and specific algorithms to maintain parallel class hierarchies [17].
Another approach is to use only concepts that already exist in the data model. This implies an implicit definition of object representations. It elaborates on existing notions, e.g., specialization, aggregates and multiple instantiation. There is no need for specific operators to define and manipulate the representations. This greatly simplifies the concepts and mechanisms involved in the implementation of the data model. This approach is used for SHOOD [23, 28]. It uses the notion of multiple instantiation, by which an instance is allowed to belong simultaneously to several classes. They are not necessarily related by a specialization relationship. An instance can therefore belong simultaneously to the Aircraft class and to the Museum_collection class (Figure 5).

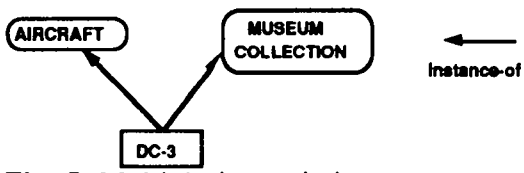


**Fig. 5.** Multiple instantiation.

This approach has the fundamental property of preserving object identity throughout representations. It is complemented by a classification mechanism called MIC - an acronym for multiple instantiation and classification.
In SHOOD, representations are objects themselves. They are modeled by classes. This provides a consistent framework in which the reflexive architecture of the model is fulfilled.
The use of multiple instantiation for multiple object representations is addressed in the next section (Section 4.1). Section 4.2 discusses the use of existing concepts. It focuses on specialization and aggregates. The respective merits of aggregates, specialization and multiple instantiation are then discussed (section 4.3). Section 4.4 describes a classification tool.

### 4.1 Multiple instantiation

Most object-oriented programming languages do not implement implicit instantiation [12]: instances of a class are not instances of its super-classes. This is usually dependent on the semantics of the inheritance relationship implemented: the most common form is the structural and behavioral inheritance, by which sub-classes inherit the structure and behavior of their super-class(es). Other forms of inheritance exist, among which are set inclusion and specialization. Using specialization, sub-classes may also refine the constraints imposed on attributes. With set inclusion, the sub-classes represent sets of instances that are sub-sets of their super-class(es). This last form prohibits exceptions. It also prohibits attribute redefinition, except by constraint

refinements. We assume that implicit instantiation is given for free using the set inclusion semantics of inheritance.
We are interested here in the explicit form of instantiation, i.e., instances can be simultaneously attached to classes that are not related by a specialization relationship. Multiple instantiation supports in our approach different semantics attached to the same objects. It therefore enhances the semantics of the classes because the various representations of a particular object may bear information that have - at least at first sight - little in common. For example the aircraft structural design includes the descriptions of the various fuselage sections and wings. This defines a particular aircraft representation. The computerized flight management system for the aircraft includes in turn a set of redundant computers that have nothing in common with the previous representation, except that, when flying, the computers output signals that will drive the aircraft controls through actuators. Note that these actuators are usually hydraulic or electric systems that are not described in either of the representations. There is therefore no explicit requirement for both representations to be related.

### 4.2 Specialization and aggregates

A crude approach to the implementation of multiple object representations is to define a common sub-class that will hold the corresponding instances. The super-classes of this inter-representation class are the classes that define the various representations. If implicit instantiation is available, this also preserves object identity. Another advantage is that this approach allows an easy location of the semantic relationships existing between the different representations, i.e., in the common inter-representation class. This simplifies somewhat the class graph. In the example, an inter-representation class called Inter_Rep_Aircraft is defined as a common sub-class for the classes Structure and Flight_Management. It holds the constraints and relationships connecting both representations (Figure 6).

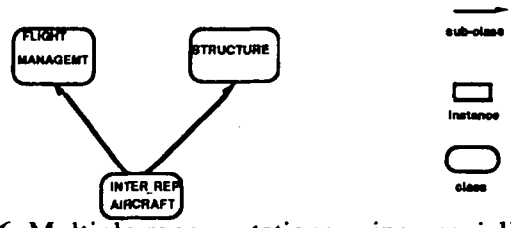


**Fig. 6.** Multiple representations using specialization.

It is our opinion that this is a misuse of the specialization relationship. An aircraft is not a sub-class of its structural or electronic representations. These should be better defined as components of the aircraft. The latter is a large composite object that includes, among others, a structural part and lots of avionics sub-parts. If available, this can be implemented by a composition relationship (Figure 7). Aggregates of components do not preserve object identity. The aggregate classes define inter-representation classes, e.g., the class Aircraft in Figure 7. Each of their attributes models a specific representation, e.g., the attributes f and s. The domains of these attributes are the classes that define each representation, e.g., Flight_management and Structure. An object instance X that belongs to the aggregate class is therefore an aggregate of references to the particular instances Y, Z, ... of the representations. These references are object identifiers. Aggregates cannot therefore implement object identity.

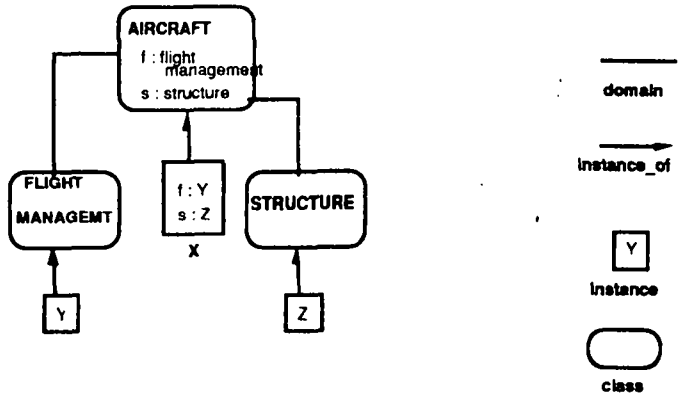


**Fig. 7.** Multiple representations using aggregates.

Nonetheless, aggregates bear specific advantages. First, the various representations may be defined independently of each other. This advantage is also obvious with multiple instantiation. The class graph can therefore be made simpler and is easier to maintain. This departs from the specialization approach, where sub-classes can be defined to refine the super-classes (the standard usage of specialization) but also to group representations. Here, the sub-class is an inter-representation class (Figure 8). This is confusing.
It is our opinion that this approach should be avoided. In contrast, aggregates can be used while keeping the original semantics of specialization, i.e., refining super-classes. This has an interesting impact on the effectiveness of classification mechanisms. The simpler the specialization graph, the better the classification efficiency, because it is not disturbed by the inter-representation classes. This is discussed in Section 4.4.

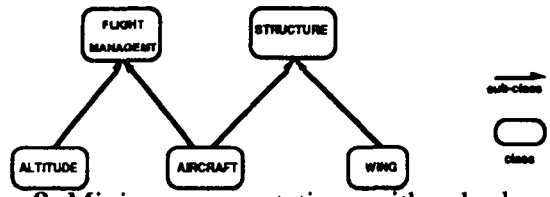


**Fig. 8.** Mixing representations with sub-classes.

### 4.3 Aggregates and multiple instantiation

As mentioned above, multiple instantiation is effective when the representations have no explicit semantic relationships. This seems counter-intuitive, but the example in Section 4.1 gives such an example. It is also interesting if the users focus on individual instances and not on whole classes. This is usually the case in engineering design environments.
If either of these requirements is not met, aggregates can be combined in SHOOD with multiple instantiation. The existence of inapplicable representations is then modeled by the type constructor "OR". If type A and type B are defined, the type "A OR B" denotes the union of the sets of instances of type A and of type B. An aircraft is the union of its structural and electronic representations: "electronic OR structural". But a commercial aircraft, e.g., "a1", has both structural and electronic representations. It is therefore defined by the type "electronic AND structural". This means that both representations are mandatory for this type of aircraft. This approach has the advantage of preserving the specialization and inheritance semantics in SHOOD because the set of aircraft with type "A AND B" is a subset of the set of aircraft of type "A OR B".
It is possible to define inter-representation relationships and constraints in the aggregates, e.g., the total weight of the aircraft is the sum of the weights of each part in the various representations and it cannot exceed X tons.
Combination of aggregates and multiple instantiation has the following advantages:

- object identity is preserved,
- multiple representations can be worked out in parallel,
- sequencing the design is expressible among representations,
- consistency within representations is standard,
- consistency among representations is expressible and precisely located, i.e., in the aggregates.

### 4.4 Classification

The rationale for design applications is to define precisely artifacts, starting with general user requirements and refining the object until a complete and consistent model of the object is characterized. The model and the values evolve continuously. Flexible instantiation is potentially a powerful tool to take these evolutions into account. A frequent need of the designers is also the requirement for automated classification mechanisms. The design object may reach a particular design phase and various alternatives may be available to resume the process. Characterizing the classes to which a particular object resembles most can therefore be a valuable tool to meet some design constraints if experience already exists for similar objects. This stresses the increasing role of classification mechanisms. They yield two results:

- refinement in the knowledge pertaining to the objects,
- modification of existing knowledge.

This is described in the next section.

**Object migration.** Refinement in the knowledge corresponding to the objects is achieved by attaching them to new specialized classes. For example, incomplete objects may see the addition of attribute values that enable the designers to attach them to sub-classes that are more detailed: if "my_aircraft" is an instance of the classes Recreational and Home-built, and if it is now fitted with floats, it can now be attached to the class Amphibian (Figure 9).

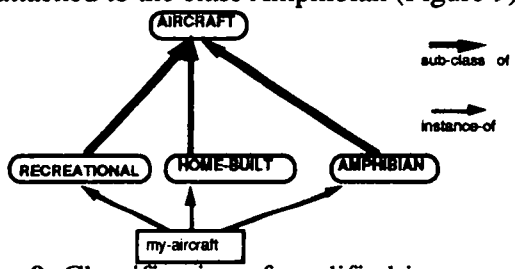


**Fig. 9.** Classification of modified instances.

If the attribute values change in the objects, the classes to which they belong may change. For example, my DC-3 used to belong to the Commercial aircraft class only. Being too old now, it has to migrate to the class Collection_item.

A similar requirement exists when the class graph is modified [5]. This issue is often called schema evolution [20]. If the designers require the addition or deletion of attributes in the classes or relationships between classes, the existing instances may be required to migrate because they do not conform to the new class definitions anymore. For example, if the concept of business aircraft evolves from aircraft with at most 12 passengers to aircraft carrying 8 passengers at most, all the business aircraft with 9 to 12 passengers must migrate from the class Business to the class Commercial (Figure 10).

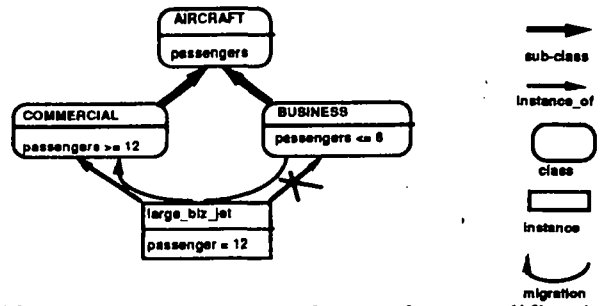


**Fig. 10.** Migrating instances due to class modifications.

**Creating objects and representations.** On the one hand, classification helps considering object evolution. On the other hand, multiple instantiation supports multiple object representations. Attaching objects to classes is the result of a classification process. This is true even when creating the objects: it is first questionable whether a given instance, when just created, conforms to any class definition. A classification process must tell whether it can be an instance of the specified class(es). If it can, the instantiation mechanism implements the "instance_of" relationship between the instance and the corresponding class(es). In SHOOD, creating an instance is therefore amenable to:

- a tentative classification in the classes where it is created: if cleared, the classes are qualified "potential",
- the creation of an "instance_of" relationship between the instance and the class(es).

Refinement of the knowledge pertaining to an object X in class C may then be implemented by the tentative classification of X in the sub-classes of C. As above for object creation and representation creation, it suffices to find out the sub-classes of C that are potential classes and attach X to some of them.

In existing object-oriented environments, the classification process is usually encoded in the appropriate creation methods in a degenerated form. It takes the form of a type checking algorithm. The latter is most often implemented as a component of a larger type-checking system. Although appropriate in programming languages, this approach is inadequate for knowledge representation. In this context, classification mechanisms are specifically designed to monitor evolving objects that may migrate from one class to another, due to changing values or definitions. This contrasts dramatically with programming languages where objects are designed primarily to be stored in shared libraries of stable components that are subsequently reused by multiple users. The requirement for object stability is therefore a first concern. As mentioned in Section 1, our starting point is the design of an object-based knowledge representation model. So, we do not necessarily conform to the requirements of object-oriented programming environments, or software engineering by large. In particular, we depart from the stability requirement.

Generalizing the classification to multiple representations, we consider the creation of an object instance as:

- a classification of the instance in the class(es) specified when it is created,
- the existence of a particular representation for the object, i.e., its creation class.

Creating a new representation for an object instance requires a subsequent classification:

- the new representation must be a potential class,
- if cleared, another "instance_of" relationship can be implemented between the object and the new representation. Note that this does not require the various representations to be related by any specialization relationship.

Classification mechanisms allow therefore the following statements to be fully supported:

- let my DC-3 be an instance of the classes "Aircraft" and "Collection" simultaneously,
- attach my DC-3 to the most specific sub-class of Aircraft,
- if my DC-3 is painted in red, make it an instance of the "Red_aircraft" class, which is a sub-class of Aircraft. My DC-3 will subsequently be an instance of the classes "Red_aircraft" and "Collection" simultaneously.

Classification mechanisms and multiple instantiation are therefore unifying concepts that are used here to implement such different operations as:

- object creation and modification,
- representation creation and modifications.

## 5 Perspectives

### 5.1 Implementation

The model SHOOD is implemented in Le_Lisp™ and Aida™ on Sun SPARCstation 2 ™ running SunOS, version 4.1.1, and X-Window, release X11/R4. Due to its reflexive architecture, the model is generated using a small meta-class bootstrap. The code includes approximately 4,000 lines of code. The user interface is based on a menu-driven interaction with the designer. The main window gives the opportunity to visualize the inheritance and the instantiation graphs. They can be selectively browsed to scan particular sub-graphs. Class can then be edited and modified dynamically. Methods can similarly be defined, stored and executed. Particular menu sequences allow for class modifications with delayed updates of the instances involved. Instances can also be manipulated and modified after direct selection in the instantiation graph in the main window. Multiple instantiation is also directly available and updatable in the main window. A menu bar above the main window provides buttons for:

- database creation, saving, and loading,
- class, attribute, method, inference and constraint creation,
- inheritance or instantiation graph browsing,
- class editing,

using specific pop-up menus. They allow interactive, i.e., mouse-driven, selection of parameters (types, domains, etc.) for the manipulation and creation of classes, instances, etc. Connection with a 3D geometric modeler is underway.

### 5.2 Research issues

**Persistence.** A primitive form of persistence is available through a button in the main window. The user can save dynamically his work during a session and reload it later during another session. The secondary storage houses an intermediate form of code that can be saved and reloaded very rapidly because it is directly interpreted by the system. Several hundred classes and instances can be saved or loaded in less than a second. The user cannot so far load several databases simultaneously. An alternative to persistence that is explored is to extend the meta-object kernel in a way similar to PCLOS [24]. It defines persistent classes as instances of meta-classes that are specializations of the original (non-persistent) meta-classes in the kernel, with ad-hoc methods attached. In SHOOD so far, transient (temporary) classes are not explicitly provided. Extensions to persistence will include the implementation of transactions, with undo and redo facilities. Also planned in the future is an enhanced multi-user version of the system.

**Rules.** Still the subject of active research is the definition and integration of a declarative form of object manipulation based on rules [3, 16]. We believe that an implementation of rules will outperform many programming paradigms when using SHOOD. This is because rules will allow consistency checking to be defined simply. It will also allow scripts to be defined for method combination (see next sub-section), for inference selection in attribute computation, and will itself be used for rule triggering using a meta-rule programming approach. A first proposal is based on rules defined by <event, condition, action> triplets. It will benefit from the modeling power of SHOOD, i.e., we believe like others that "*rules are objects too*" [7].

**Method combination.** A promising side-effect of method management in a specific graph is the opportunity for advanced method combination. As defined in Section 3.3, methods are

implemented by classes forming a separate method graph. Selecting the appropriate methods for the execution of an operation is based on all the arguments. Several eligible methods may result from the selection process. It is the user's responsibility to define the order in which these selected methods must be executed. The system will otherwise execute them in a random order with potentially undesirable side-effects. Method combination will allow the user to select the order of method executions and possibly to exclude potential conflicts by specifying ad-hoc control statements. This area is being actively explored. A default scheme is presently executed for multiple selection, by which the first available result is returned, exclusive of any other.

## 6 Conclusion

The model SHOOD presented in this paper is designed for highly dynamic applications, i.e., for which the knowledge on the application domain may evolve. This means that partial pieces of information must be integrated, new relationships between objects may be discovered long after their first encounter, new unexpected attributes may appear for specific unhypothesized characteristics of the objects and so on. Engineering design and genetics are examples of such applications. Usually, redesign of existing artifacts is practiced as a refinement process. But this is ineffective. This departs dramatically from previous assumptions in the literature.
To support this dynamicity, SHOOD provides features that are seldom found simultaneously in a single and unifying framework. It elaborates on object-oriented paradigms and knowledge representation languages. Among the features supported are:

- complex composite objects,
- user-defined semantic relationships,
- incompletely defined or partially inconsistent objects,
- multiple object representations,
- multiple inferencing,
- a powerful although flexible method mechanism,
- a classification mechanism,
- object persistence.

The model is implemented with a meta-class kernel that defines all the basic concepts and provides the features required to implement the application level. It is also the basis for the extensibility required by the applications. New concepts and new relationships can be implemented by definition of ad-hoc meta-classes holding the appropriate functionalities, i.e., attributes and methods. They can later be instantiated to provide the user with the desired objects.
This proposal is based on experience gained in mechanical CAD/CAM for GSIP (Groupement Scientifique Interdisciplinaire de Productique). It is a joint research effort in Computer Integrated Manufacturing from the Universities of Grenoble with industrial participants.

## Acknowledgements

The authors thank J.M BONNEFOND, F. BOUNAAS from IMAG, and M. TOLLENAERE from Institut de Mécanique de Grenoble (Laboratoire 3S) for their contributions to this work. It is supported in part by INRIA for the project SHERPA and by GSIP.

## References


1. Ahmed R., Navathe S. Version management of composite objects in CAD databases. Proc. ACM SIGMOD '91 Conf. Denver (Co). 1991.
2. Banerjee J., Kim W., Kim K.J., Korth H. Semantics and implementations of schema evolution in object-oriented databases. Proc. ACM SIGMOD '87 Conf. San Francisco (Ca). 1987.
3. Bauzer-Medeiros C., Pfeffer P. Object integrity using rules. Proc. ECOOP '91.Geneva (CH). July 1991.
4. Carré B., Geib J.M. The point of view notion for multiple inheritance. Proc. ECOOP/OOPSLA '90. Ottawa (C). 1990.
5. Casais E. Managing class evolution in object-oriented systems. Centre Universitaire Informatique. Université de Genève (CH). 1990.
6. Cointe P. Metaclasses are First Class: the ObjVlisp Model. ProceedingsOOPSLA'87. 1987.
7. Dayal U. & al. Rules are objects too: a knowledge model for an active object- oriented database system. Advances in object-oriented database systems. Bad-Munster (G). September 1988.
8. Escamilla J., Jean P. Relationships in an Object Knowledge Representation Model. Proc. 2nd Intl. Conf. Tools for Artificial Intelligence. Washington (D.C). 1990.
9. Fox M., Wright JM., Adam D. Experiences with SRL: an analysis of a frame-based knowledge representation. Proc. 1st Intl. Conf. Expert Database Systems.Charleston (SC). 1986.
10. Gabriel R.P & al. CLOS: integrating object-oriented and functional programming. Comm. ACM. **34**(9). 1991.
11. Giacometti F., Chang T.C. Object-oriented design for modelling parts, assemblies and tolerances. Proc. 2nd Intl. Conf. TOOLS '90. Paris (F). 1990.
12. Goldberg A., Robson D. Smalltalk 80: the language and its implementation. Addison-Wesley. 1983.
13. Keene S.E. Object-oriented programming in Common Lisp. A programmer's guide to CLOS. Addison-Wesley. 1989.
14. Keller T. & al. Efficient assembly of complex objects. Proc. ACM SIGMOD '91 Conf. Denver (Co).1991.
15. Kim W. Object-oriented databases: definition and research directions. IEEE Trans. on Knowledge and Data Engineering. **2**(3). 1990.
16. Kotz & al. Supporting Semantic Rules by a Generalized Event/Trigger Mechanism. Advances in DB technology. EDBT '88. Venice (I). March 1988.
17. Marino O. & al. Multiple perspectives and classification mechanism in object riented representation. Proc. ECAI Conf. Stockholm (S). 1990.
18. Meyer B. Object-oriented software construction. Prentice-Hall.
19. Minsky M. 1975. A framework for representing knowledge. The psychology of computer vision. McGraw-Hill. 1988.
20. Nguyen G.T., Rieu D. Schema evolution in object-oriented database systems. Data & Knowledge Engineering. North-Holland. **4**(1).1989.
21. Nguyen G.T, Rieu D. Heuristic control on dynamic database objects. Information Processing '90. Meersman (ed.). North-Holland. 1990.
22. Nguyen G.T, Rieu D. Representing design objects. Proc.1st Intl. Conf. Artificial Intelligence in Design. Edinburgh (U.K). 1991.
23. Nguyen G.T, Rieu D. Multiple object representations. Proc. 20th ACM Computer Science Conference. Kansas City (Mo). 1992.
24. Paepcke A. PCLOS: stress testing CLOS. Experiencing the metaobject protocol. Proc. ECOOP/OOPSLA '90. Ottawa (C). September 1990.
25. Pernici B. Objects with roles. Proc. Intl. Conf. on Office Information Systems. Boston (Ma). 1990.
26. Richardson J., Schwarz P. Aspects: extending objects to support multiple, independent roles. Proc ACM SIGMOD '91 Conf. Denver (Co.). 1991.

27. Rieu D., Nguyen G.T. Semantics of CAD Objects for Generalized Databases. Proc. 23rd Design Automation Conference. Las Vegas (Nevada). 1986.
28. Rieu D., Nguyen G.T. Object views for engineering databases. Proc. 3rd Intl. Conf. Data & Knowledge Systems for Manufacturing & Engineering. Lyon (F). 1992.
29. Sciore E.Object specialization. ACM Trans. on Office Information Systems. 7(2).1989.
30. Silberschatz A., Stonebraker M., Ullman J.D. Database systems: achievements and opportunities. Laguna Beach Report. TR 90-22. Dept. of Comp. Sc. The University of Texas at Austin. Austin (Tx). 1990.
31. Unland R., Schlageter G. Object-oriented database systems: concepts and perspectives. Lecture Notes in Computer Science. Springer-Verlag. 1990.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399